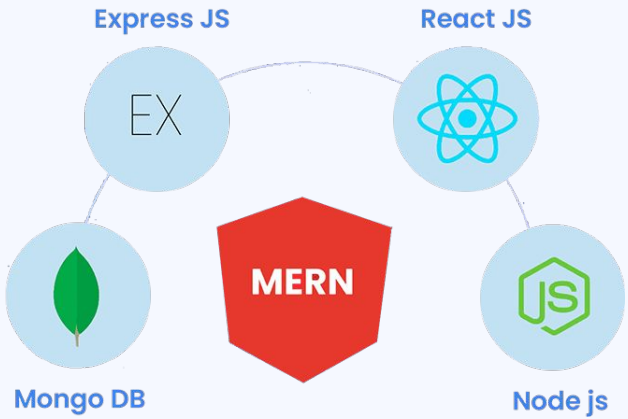


Full Stack Development Using MERN

Day 06 - 09th June 2024

Lecturer Details:

Chanaka Wickramasinghe
cmw@ucsc.cmb.ac.lk
0771570227

Anushka Vithanage
dad@ucsc.cmb.ac.lk
071 527 9016

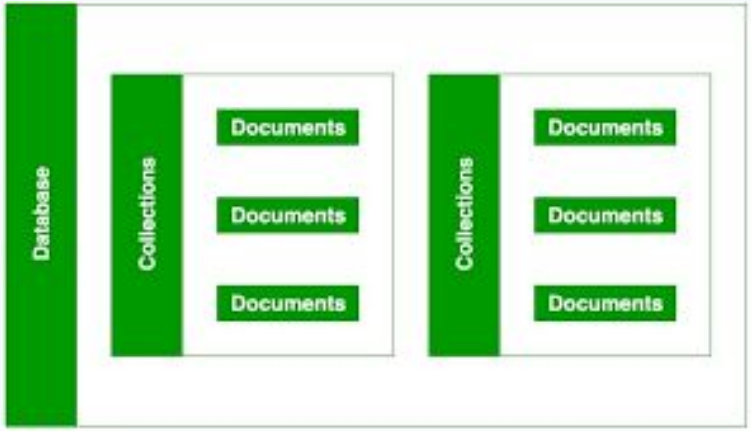
Prameeth Maduwantha
bap@ucsc.cmb.ac.lk
0772888098

Lecture 06

Data Persistence with MongoDB and Mongoose

What is MongoDB?

- MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability.
- MongoDB works on concept of collection and document.

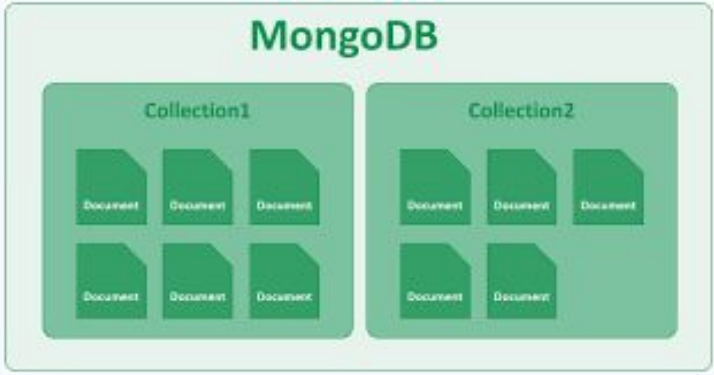


What is MongoDB?

Database

- Database is a physical container for collections.
- Each database gets its own set of files on the file system.
- A single MongoDB server typically has multiple databases.

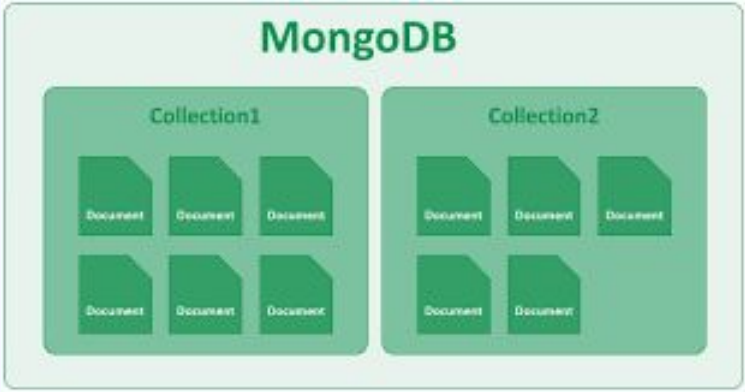
What is MongoDB?



Collection

- Collection is a group of MongoDB documents.
- It is the equivalent of an RDBMS table.
- A collection exists within a single database.
- Collections do not enforce a schema.
- Documents within a collection can have different fields.
- Typically, all documents in a collection are of similar or related purpose.

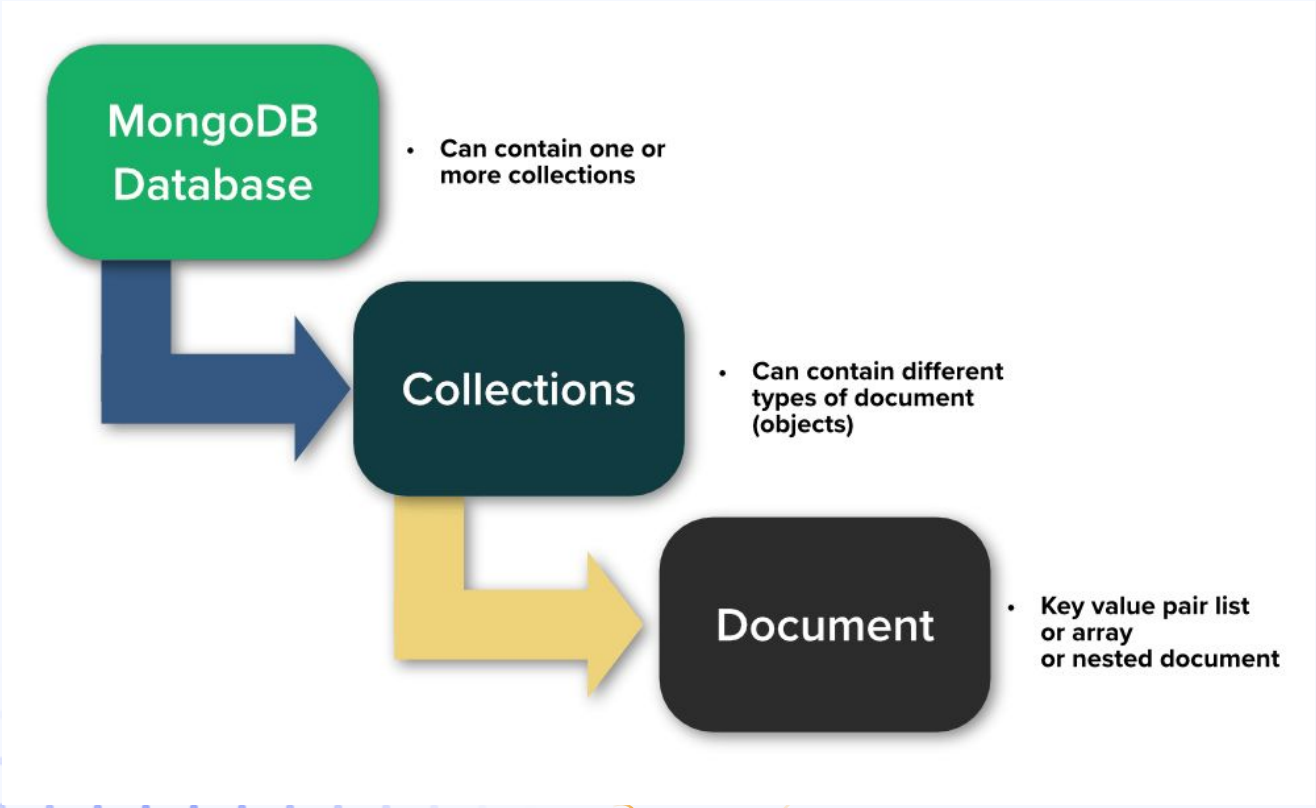
What is MongoDB?



Document

- A document is a set of key-value pairs.
- Documents have dynamic schema.
- Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

MongoDB


MongoDB Database
• Can contain one or more collections
Collections
• Can contain different types of document (objects)
Document
• Key value pair list or array or nested document

MongoDB Terminology

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by MongoDB itself)

Sample Document

_id holds a unique 12 bytes hexadecimal value for each document

```
{
   _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   by: 'tutorials point',
   url: 'http://www.tutorialspoint.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100,
   comments: [
      {
         user:'user1',
         message: 'My first comment',
         dateCreated: new Date(2011,1,20,2,15),
         like: 0
      },
      {
         user:'user2',
         message: 'My second comments',
         dateCreated: new Date(2011,1,25,7,45),
         like: 5
      }
   ]
}
```

Advantages of MongoDB over RDBMS

- **Schema less –** MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- **Structure of a single object is clear.**
- **No complex joins.**
- **Deep query-ability. –** MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- **Ease of scale-out** - MongoDB is easy to scale.

Why Use MongoDB?

- Document Oriented Storage - Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication and high availability
- Auto-Sharding
- Rich queries
- Fast in-place updates
- Professional support by MongoDB

Where to Use MongoDB?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

Install MongoDB On Windows

- To install MongoDB on Windows, first download the latest release of MongoDB from https://www.mongodb.com/download-center.
- Create an account and verify the account.
- Create a cluster.
- Connect the cluster with mongosh (Windows Shell)
  - Download the .zip file for shell
  - Extract the .zip file
  - Add to PATH (system variables)
  - `mongosh --help` to check whether its working.

MongoDB - Data Modelling

- Data in MongoDB has a flexible schema.
- They do not need to have the same set of fields or structure.
- Common fields in a collection's documents may hold different types of data.

Data Model Design

- MongoDB provides two types of data models:
  - Embedded Data Model
  - Normalized Data Model

Embedded Data Model

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

```
{
        _id: ,
        Emp_ID: "10025AE336"
        Personal_details:{
                First_Name: "Radhika",
                Last_Name: "Sharma",
                Date_Of_Birth: "1995-09-26"
        },
        Contact: {
                e-mail: "radhika_sharma.123@gmail.com",
                phone: "9848022338"
        },
        Address: {
                city: "Hyderabad",
                Area: "Madapur",
                State: "Telangana"
        }
}
```

Normalized Data Model

In this model, you can refer the sub documents in the original document, using references.

Employee:

```
{
        _id: <ObjectId101>,
        Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
        _id: <ObjectId102>,
        empDocID: " ObjectId101",
        First_Name: "Radhika",
        Last_Name: "Sharma",
        Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
        _id: <ObjectId103>,
        empDocID: " ObjectId101",
        e-mail: "radhika_sharma.123@gmail.com",
        phone: "9848022338"
}
```

Considerations while designing Schema in MongoDB

- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Create Database

use Command

- MongoDB **use DATABASE_NAME** is used to create database.
- The command will create a new database if it doesn't exist, otherwise it will return the existing database.

- Syntax:

```
use DATABASE_NAME
```

- Example:

```
>use mydb
switched to db mydb
```

Create Database

To check your currently selected database, use the command **db**

```
>db
mydb
```

If you want to check your databases list, use the command **show dbs**.

```
>show dbs
local    0.78125GB
test     0.23012GB
```

Create Database

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

```
>db.movie.insert({"name":"tutorials point"})
>show dbs
local      0.78125GB
mydb       0.23012GB
test       0.23012GB
```

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

Drop Database

MongoDB **db.dropDatabase()** command is used to drop a existing database.

Syntax:

```
db.dropDatabase()
```

If you want to delete new database <mydb>, then dropDatabase() command would be as follows.

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

Create Collection

Syntax: `db.createCollection(name, options)`

In the command, **name** is name of collection to be created.

Options is a document and is used to specify configuration of collection.

Options parameter is optional, so you need to specify only the name of the collection.

Create Collection

Following is the list of options you can use.

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. **If you specify true, you need to specify size parameter also.**
autoIndexId	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.**
max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

Create Collection

Basic syntax of **createCollection()** method **without options**

```
>use test
switched to db test
>db.createCollection("mycollection")
{ "ok" : 1 }
>
```

Create Collection (with options)

Basic syntax of **createCollection()** method **with options**

```
db.createCollection("mycol", {
    capped: true,
    size: 6142800,
    max: 10000
});
```

Create Collection

You can check the created collection by using the command **show collections**.

```
>show collections
mycollection
system.indexes
```

```
>show collections
mycol
mycollection
system.indexes
tutorialspoint
>
```

In MongoDB, you don't need to create collection. MongoDB creates collection automatically, when you insert some document.

```
>db.tutorialspoint.insert({"name" : "tutorialspoint"}),
WriteResult({ "nInserted" : 1 })
```

Drop Database

MongoDB's **db.collection.drop()** is used to drop a collection from the database.

Syntax:

```
db.COLLECTION_NAME.drop()
```

First, check the available collections into your database mydb. Now drop the collection with the name mycollection.

```
>db.mycollection.drop()
true
>
```

Datatypes supported by MongoDB

String - This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.

Integer - This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.

Boolean - This type is used to store a boolean (true/ false) value.

Double - This type is used to store floating point values.

Min/ Max keys - This type is used to compare a value against the lowest and highest BSON elements.

Datatypes supported by MongoDB

Arrays - This type is used to store arrays or list or multiple values into one key.

Timestamp - This can be handy for recording when a document has been modified or added.

Object - This datatype is used for embedded documents.

Null - This type is used to store a Null value.

Symbol - This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.

Datatypes supported by MongoDB

Date - This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.

Object ID - This datatype is used to store the document's ID.

Binary data - This data type is used to store binary data.

Code - This datatype is used to store JavaScript code into the document.

Regular expression - This datatype is used to store regular expression.

Insert Method

To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()** method.

Syntax:

```
>db.COLLECTION_NAME.insert(document)
```

```
> db.users.insert({
... _id : ObjectId("507f191e810c19729de860ea"),
... title: "MongoDB Overview",
... description: "MongoDB is no sql database",
... by: "tutorials point",
... url: "http://www.tutorialspoint.com",
... tags: ['mongodb', 'database', 'NoSQL'],
... likes: 100
... })
```

Insert Method

You can also pass an array of documents into the insert() method

```
> db.createCollection("post")
> db.post.insert([
        {
                title: "MongoDB Overview",
                description: "MongoDB is no SQL database",
                by: "tutorials point",
                url: "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 100
        },
        {
        title: "NoSQL Database",
        description: "NoSQL database doesn't have tables",
        by: "tutorials point",
        url: "http://www.tutorialspoint.com",
        tags: ["mongodb", "database", "NoSQL"],
        likes: 20,
        comments: [
                {
                        user:"user1",
                        message: "My first comment",
                        dateCreated: new Date(2013,11,10,2,35),
                        like: 0
                }
        ]
}
])
```

The insertOne() method

If you need to insert **only one document** into a collection you can use this method.

Syntax:

```
>db.COLLECTION_NAME.insertOne(document)
```

```
> db.empDetails.insertOne(
        {
                First_Name: "Radhika",
                Last_Name: "Sharma",
                Date_Of_Birth: "1995-09-26",
                e_mail: "radhika_sharma.123@gmail.com",
                phone: "9848022338"
        })
```

The `insertMany()` method

You can insert **multiple documents** using the insertMany() method. To this method you need to pass an array of documents.

```
> db.empDetails.insertMany(
        [
                {
                        First_Name: "Radhika",
                        Last_Name: "Sharma",
                        Date_Of_Birth: "1995-09-26",
                        e_mail: "radhika_sharma.123@gmail.com",
                        phone: "9000012345"
                },
                {
                        First_Name: "Rachel",
                        Last_Name: "Christopher",
                        Date_Of_Birth: "1990-02-16",
                        e_mail: "Rachel_Christopher.123@gmail.com",
                        phone: "9000054321"
                },
                {
                        First_Name: "Fathima",
                        Last_Name: "Sheik",
                        Date_Of_Birth: "1990-02-16",
                        e_mail: "Fathima_Sheik.123@gmail.com",
                        phone: "9000054321"
                }
        ]
)
```

The find() Method

To query data from MongoDB collection, you need to use MongoDB's find() method.

Syntax: `>db.COLLECTION_NAME.find()`

```
> db.mycol.find()
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534c"), "title" : "MongoDB Overv
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534d"), "title" : "NoSQL Databas
>
```

To display the results in a formatted way, you can use **pretty()** method.

The findOne() method

Apart from the find() method, there is findOne() method, that returns only one document.

```
>db.COLLECTIONNAME.findOne()
```

```
> db.mycol.findOne({title: "MongoDB Overview"})
{
        "_id" : ObjectId("5dd6542170fb13eec3963bf0"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
}
```

RDBMS Where Clause Equivalents in MongoDB

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:{$eg;<value>}}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50

Operation	Syntax	Example	RDBMS Equivalent
Greater Than Equals	{<key>: {$gte: <value>}}	db.mycol.find({"likes": {$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>: {$ne: <value>}}	db.mycol.find({"likes": {$ne:50}}).pretty()	where likes != 50
Values in an array	{<key>: {$in: [<value1>, <value2>,..... <valueN>]}}	db.mycol.find({"name":{$in: ["Raj", "Ram", "Raghu"]}}).pretty()	Where name matches any of the value in :["Raj", "Ram", "Raghu"]
Values not in an array	{<key>: {$nin: <value>}}	db.mycol.find({"name": {$nin:["Ramu", "Raghav"]}}).pretty()	Where name values is not in the array : ["Ramu", "Raghav"] or, doesn't exist at all

AND, OR & NOT

Age = 25 **AND** City = “Colombo”

Age = 25 **OR** City = “Colombo”

Age **NOT** Less Than 25

AND, OR & NOT

Age = 25 **AND** City = “Colombo”

- Checks both, Age should be equal to 25 and City should be Colombo

Age = 25 **OR** City = “Colombo”

- Either Age can be 25 or City can be Colombo

Age **NOT** Less Than 25

- Age should be not less than 25, i.e. Age should be greater than or equal to 25.

AND, OR & NOT

AND –

```
>db.mycol.find({ $and: [ {<key1>:<value1>}, { <key2>:<value2>} ] })
```

OR –

```
>db.mycol.find(
  {
    $or: [
      {key1: value1}, {key2:value2}
    ]
  }
).pretty()
```

NOT –

```
>db.COLLECTION_NAME.find(
      {
            $not: [
                  {key1: value1}, {key2:value2}
            ]
      }
)
```

AND, OR & NOT

```
db.mycol.find({$and:[{"by":"a1"},{"title": "t1"]}])

db.mycol.find({$or:[{"by":"a2"},{"title": "t2"]}])

db.mycol.find({"likes": {$gt:10}, $or: [{"by": "a3"}, {"title": "t3"}]})

db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )
```

Update() Method

The update() method updates the values in the existing document.

Syntax: `>db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)`

```
db.mycol.update({'title':'MongoDB'},{$set:{'title':'New MongoDB'}})
```

By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

```
>db.mycol.update({'title':'MongoDB Overview'},
   {$set:{'title':'New MongoDB Tutorial'}},{multi:true})
```

findOneAndUpdate() method

The findOneAndUpdate() method updates the values in the existing document.

Syntax: >db.COLLECTION_NAME.findOneAndUpdate(SELECTIOIN_CRITERIA, UPDATED_DATA)

```
> db.empDetails.findOneAndUpdate(
        {First_Name: 'Radhika'},
        { $set: { Age: '30',e_mail: 'radhika_newemail@gmail.com'}}
)
```

updateOne() method

This methods updates a single document which matches the given filter.

Syntax:

```
>db.COLLECTION_NAME.updateOne(<filter>, <update>)
```

```
> db.empDetails.updateOne(
        {First_Name: 'Radhika'},
        { $set: { Age: '30',e_mail: 'radhika_newemail@gmail.com'}}
)
```

updateMany() method

The updateMany() method updates all the documents that matches the given filter.

Syntax:

```
>db.COLLECTION_NAME.update(<filter>, <update>)
```

```
> db.empDetails.updateMany(
        {Age:{ $gt: "25" }},
        { $set: { Age: '00'}}
)
```

The `remove()` Method ctd.

remove() method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

- **deletion criteria** - (Optional) deletion criteria according to documents will be removed.
- **justOne** - (Optional) if set to true or 1, then remove only one document.

The remove() Method

```
>db.mycol.remove({'title':'MongoDB Overview'})

WriteResult({"nRemoved" : 1})

> db.mycol.find()

{"_id" : ObjectId("507f191e810c19729de860e2"), "title" : "NoSQL Overview" }

{"_id" : ObjectId("507f191e810c19729de860e3"), "title" : "Tutorials Point
Overview" }
```

Projection

- In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document.
- If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

Projection - find() method

- find() method, explained in MongoDB Query Document accepts second optional parameter that is list of fields that you want to retrieve.
- In MongoDB, when you execute find() method, then it displays all fields of a document. *To limit this, you need to set a list of fields with value 1 or 0*.

  1 is used to show the field while 0 is used to hide the fields.

Projection - find() method

{_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"},
{_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"},
{_id : ObjectId("507f191e810c19729de860e3"), title: "Tutorials Point Overview"}

```
>db.mycol.find({},{"title":1,_id:0})
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
{"title":"Tutorials Point Overview"}
>
```

The Limit() Method

To limit the records in MongoDB, you need to use **limit()** method. The method accepts one number type argument, which is the number of documents that you want to be displayed.

Syntax:

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```

```
>db.mycol.find({},{"title":1,_id:0}).limit(2)
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
>
```

The skip() Method

Apart from **limit()** method, there is one more method skip() which also accepts number type argument and is used to skip the number of documents.

Syntax:

```
>db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```

```
>db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)
{"title":"NoSQL Overview"}
>
```

The sort() Method

To sort documents in MongoDB, you need to use **sort()** method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

Syntax:

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})
{"title":"Tutorials Point Overview"}
{"title":"NoSQL Overview"}
{"title":"MongoDB Overview"}
>
```

Indexing

- Indexing is improves the performance of database queries by allowing the database to locate and retrieve documents more efficiently.
- Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement
- Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form.
- Index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index

Indexing Example

Let's assume we have the following documents in our "books" collection

```
[
  {
    _id: ObjectId("653bdfc9d6bbd9afdf0dbd4f"),
    title: 'Book A',
    author: 'Author X',
    publication_year: 2010,
    genre: 'Fiction'
  },
  {
    _id: ObjectId("653bdfc9d6bbd9afdf0dbd50"),
    title: 'Book B',
    author: 'Author Y',
    publication_year: 2005,
    genre: 'Non-fiction'
  },
  {
    _id: ObjectId("653bdfc9d6bbd9afdf0dbd51"),
    title: 'Book C',
    author: 'Author Z',
    publication_year: 2010,
    genre: 'Mystery'
  }
]
```

Indexing Example Contd

Let’s create an index on the "publication_year" field in a MongoDB collection named "books," you can use the createIndex method as follow.

- **db.books.createIndex({ publication_year: 1 })**

Here name of the field on which you want to create index and 1 is for ascending order and -1 for descending order used. After creating the index, MongoDB internally organizes the data as follows

Index on "publication_year":

{ 2005: [{title: "Book B", ...}],

2010: [{title: "Book A", ...}, {title: "Book C", ...}] }

Indexing Example Contd

The index itself is used by MongoDB internally to improve the speed of queries that involve filtering, sorting, or searching based on the indexed field.

For example, if you want to find books published in the year 2010, you can run a query like this

```
Atlas atlas-t2f2u1-shard-0 [primary] mydb> db.books.find({ publication_year: 2010 })
[
  {
    _id: ObjectId("653bdfc9d6bbd9afdf0dbd4f"),
    title: 'Book A',
    author: 'Author X',
    publication_year: 2010,
    genre: 'Fiction'
  },
  {
    _id: ObjectId("653bdfc9d6bbd9afdf0dbd51"),
    title: 'Book C',
    author: 'Author Z',
    publication_year: 2010,
    genre: 'Mystery'
  }
]
```

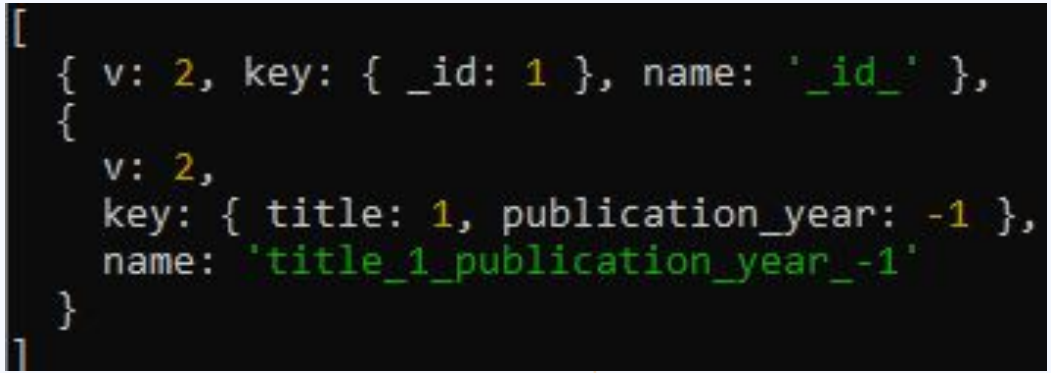
Get Indexes Created

Let's create 2 indexes in the named mycol collection as shown below

- **db.books.createIndex({"title":1,"publication_year":-1})**

Use getIndexes() method to return the indexes created.

- **db.books.getIndexes()**



Get Indexes Created Contd

In the above example

v: This field indicates the index version. In this case, it's version 2.

key: This field specifies the fields that make up the index.

name: This is the name of the index.

- **{ v: 2, key: { _id: 1 }, name: '_id_' }**
  - This first index is the default "_id" index, created automatically for every collection.
- **{ v: 2, key: { title: 1, publication_year: -1 }, name: title_1_publication_year_-1' }**
  - The second index is a compound index on the "title" and "publication_year" fields. "title" field in ascending order (1) and the "publication_year" field in descending order (-1)

Single-field Index:

- A single-field index involves indexing a single field in a document.
- Example: Creating an index on the email field. This is useful when queries often involve searching for users by their email addresses.

Compound Index:

- A compound index involves indexing multiple fields together.
- Example: Creating an index on the username and age fields together. This is useful when queries frequently involve both the username and age in the filter condition.

The key difference is that a single-field index is created on a single field, while a compound index is created on multiple fields together. Compound indexes can be beneficial for queries that involve filtering or sorting on multiple fields.

Drop Indexes

The basic syntax of DropIndex() method is as follows().

- **db.COLLECTION_NAME.dropIndex({KEY:1})**

As an example

- **db.books.dropIndexes({title:1,publication_year: 1 })**

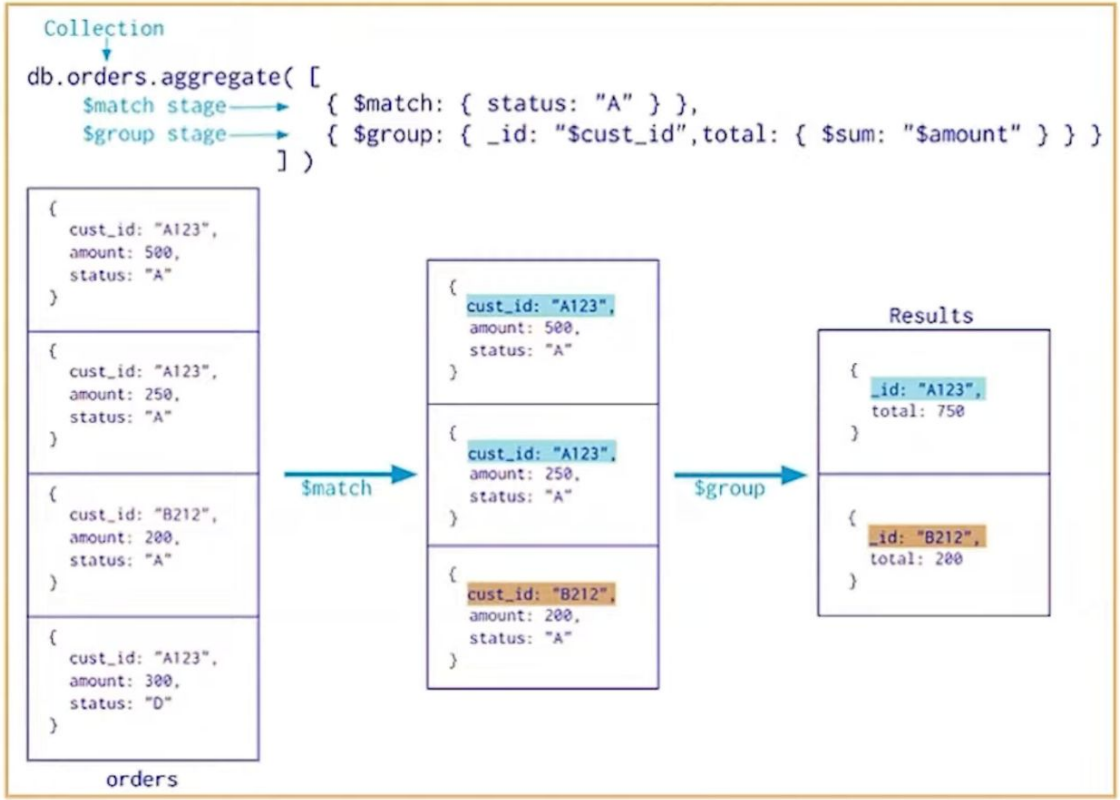
Aggregation

- Aggregation operations process data records and return computed results
- Aggregation operations, group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result
- In SQL count(*) and with group by is an equivalent of MongoDB aggregation.
- In MongoDB, the aggregation pipeline is a series of stages, where the possibility to execute an operation on some input and use the output as the input for the next command and so on .
- Documents can be filtered, transformed, sorted, or regrouped as they flow from one stage to the next

Aggregation Stages

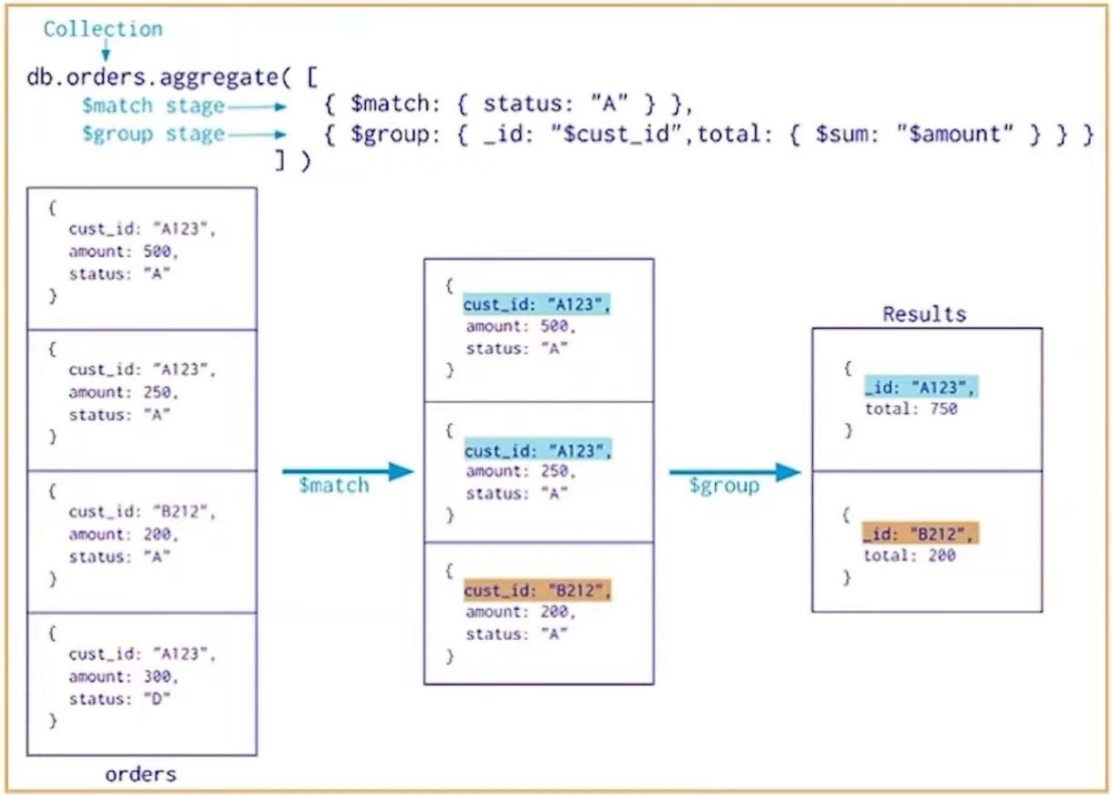
- $match: Filters documents based on a given condition.
- $group: Groups documents by a specified expression.
- $sort: Sorts documents in a defined order.
- $project: Returns only the specified fields from the document
- $addFields: This will return the documents along with a new field
- $lookup: Performs a left outer join with another collection.
- $out: This aggregation stage writes the returned documents from the aggregation pipeline to a collection.

MATCH


Collection
db.orders.aggregate([
$match stage
{ $match: { status: "A" } },
$group stage
{ $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
])
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

MATCH

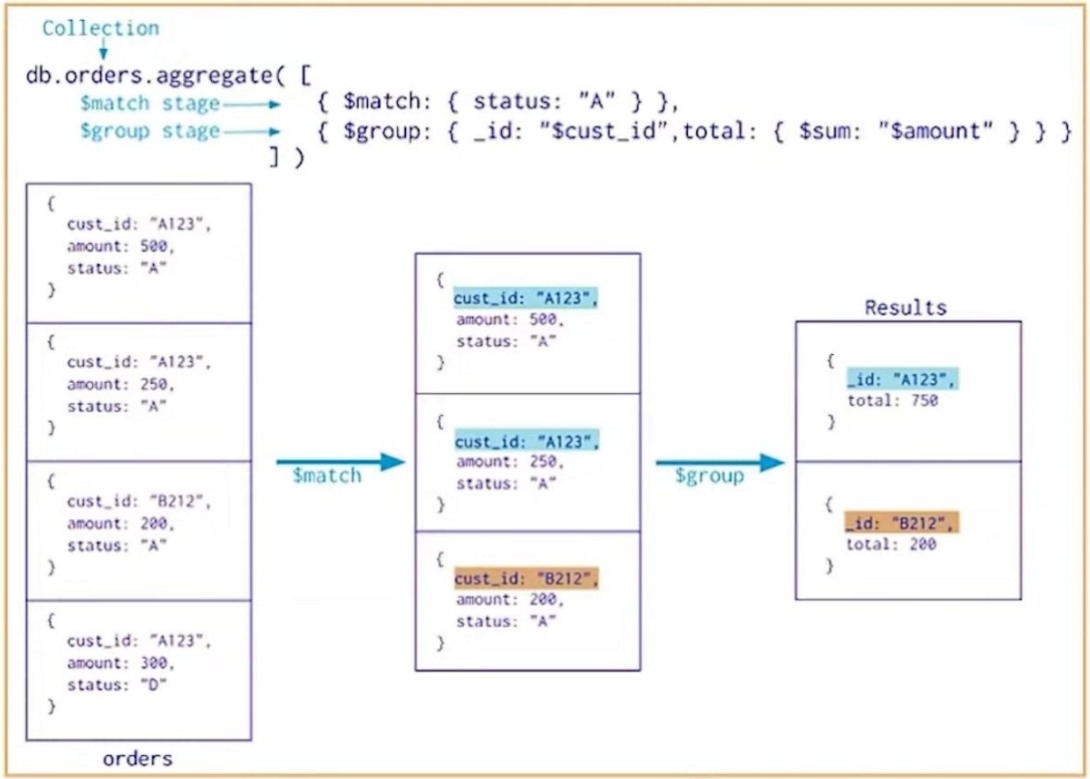
GROUP


Collection
db.orders.aggregate([
$match stage
{ $match: { status: "A" } },
$group stage
{ $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
])
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

MATCH

GROUP

SORT



Aggregation Example

Let's assume we have the following documents in our "posts" collection

```
Atlas atlas-ynt2d2-shard-0 [primary] test> db.posts.find()
[
  {
    _id: ObjectId("65fd9401b650b778080af416"),
    title: 'Post Title 2',
    body: 'Body of post.',
    category: 'Event',
    likes: 2,
    tags: [ 'news', 'events' ],
    date: 'Fri Mar 22 2024 19:51:52 GMT+0530 (India Standard Time)'
  },
  {
    _id: ObjectId("65fd9401b650b778080af417"),
    title: 'Post Title 3',
    body: 'Body of post.',
    category: 'Technology',
    likes: 3,
    tags: [ 'news', 'events' ],
    date: 'Fri Mar 22 2024 19:51:53 GMT+0530 (India Standard Time)'
  },
  {
    _id: ObjectId("65fd9401b650b778080af418"),
    title: 'Post Title 4',
    body: 'Body of post.',
    category: 'Event',
    likes: 4,
    tags: [ 'news', 'events' ],
    date: 'Fri Mar 22 2024 19:51:53 GMT+0530 (India Standard Time)'
  }
]
```

Aggregation Pipeline

1. Match and Group

```
Atlas atlas-ynt2d2-shard-0 [primary] test> db.posts.aggregate([
...   // Stage 1: Only find documents that have more than 1 like
...   {
...     $match: { likes: { $gt: 1 } }
...   },
...   // Stage 2: Group documents by category and sum each categories likes
...   {
...     $group: { _id: "$category", totalLikes: { $sum: "$likes" } }
...   }
Atlas atlas-ynt2d2-shard-0 [primary] test>
[
  { _id: 'Event', totalLikes: 6 },
  { _id: 'Technology', totalLikes: 3 }
]
```

Aggregation Example 1

- **$group:** This is an aggregation stage that groups input documents by a specified expression and outputs a document for each unique grouping.
  - **_id:** This is defining the field by which the documents will be grouped
  - **count**: For each group, we're counting the number of documents
  - **$sum**: Operator adds up the values for each group

- **$sort:** This is an aggregation stage that sorts the input documents.
  - **_id: 1:** This specifies the field by which the documents will be sorted and the sort order.

Aggregation Pipeline

2. Limit

```
Atlas atlas-ynt2d2-shard-0 [primary] test> db.posts.aggregate([
...   {
...     $match: { likes: { $gt: 1 } }
...   },
...   {
...     $group: { _id: "$category", totalLikes: { $sum: "$likes" } }
...   },
...   {
...     $limit: 1
...   }
... ])
[ { _id: 'Event', totalLikes: 6 } ]
```

Aggregation Pipeline

3. project

```
Atlas atlas-ynt2d2-shard-0 [primary] test> db.posts.aggregate([
...   {
...     $project: {
...       "title": 1,
...       "body": 1,
...       "category": 1
...     }
...   }
... ])
[
  {
    _id: ObjectId("65fd9401b650b778080af416"),
    title: 'Post Title 2',
    body: 'Body of post.',
    category: 'Event'
  },
  {
    _id: ObjectId("65fd9401b650b778080af417"),
    title: 'Post Title 3',
    body: 'Body of post.',
    category: 'Technology'
  },
  {
    _id: ObjectId("65fd9401b650b778080af418"),
    title: 'Post Title 4',
    body: 'Body of post.',
    category: 'Event'
  }
]
```

Aggregation Pipeline

4. sort

```
Atlas atlas-ynt2d2-shard-0 [primary] test> db.posts.aggregate([
...   {
...     $project: {
...       "title": 1,
...       "body": 1,
...       "likes": 1
...     }
...   },
...    {
...     $sort: {
...       "likes": -1
...     }
...    }
... ])
[
  {
    _id: ObjectId("65fd9401b650b778080af418"),
    title: 'Post Title 4',
    body: 'Body of post.',
    likes: 4
  },
  {
    _id: ObjectId("65fd9401b650b778080af417"),
    title: 'Post Title 3',
    body: 'Body of post.',
    likes: 3
  },
  {
    _id: ObjectId("65fd9401b650b778080af416"),
    title: 'Post Title 2',
    body: 'Body of post.',
    likes: 2
  }
]
```

Aggregation Pipeline

5. addFields

```
Atlas> db.posts.aggregate([
...     {
...     $addFields: {
...       isPopular: { $gt: ["$likes", 2] }
...     }
...   },
...   {
...     $project: {
...       "title": 1,
...       "body": 1,
...       "likes": 1,
...       "isPopular": 1
...     }
...   }
... ])
[
  {
    _id: ObjectId("65fd9401b650b778080af416"),
    title: 'Post Title 2',
    body: 'Body of post.',
    likes: 2,
    isPopular: false
  },
  {
    _id: ObjectId("65fd9401b650b778080af417"),
    title: 'Post Title 3',
    body: 'Body of post.',
    likes: 3,
    isPopular: true
  },
  {
    _id: ObjectId("65fd9401b650b778080af418"),
    title: 'Post Title 4',
    body: 'Body of post.',
    likes: 4,
    isPopular: true
  }
]
```

Aggregation Pipeline

6. Lookup

There are four required fields:

- from: The collection to use for lookup in the same database
- localField: The field in the primary collection that can be used as a unique identifier in the from collection.
- foreignField: The field in the from collection that can be used as a unique identifier in the primary collection.
- as: The name of the new field that will contain the matching documents from the from collection.

Lookup Example

```
Atlas atlas-ynt2d2-shard-0 [primary] test> db.posts.aggregate([
...   {
...     $lookup: {
...       from: "authors",
...       localField: "authorId",
...       foreignField: "_id",
...       as: "authorDetails"
...     }
...   }
... ])
```

```
[
  {
    _id: ObjectId("65fd9401b650b778080af416"),
    title: 'Post Title 2',
    body: 'Body of post.',
    category: 'Event',
    likes: 2,
    tags: [ 'news', 'events' ],
    date: 'Fri Mar 22 2024 19:51:52 GMT+0530 (India Standard Time)',
    authorId: 1,
    authorDetails: [ { _id: 1, name: 'Author One' } ]
  },
  {
    _id: ObjectId("65fd9401b650b778080af417"),
    title: 'Post Title 3',
    body: 'Body of post.',
    category: 'Technology',
    likes: 3,
    tags: [ 'news', 'events' ],
    date: 'Fri Mar 22 2024 19:51:53 GMT+0530 (India Standard Time)',
    authorId: 2,
    authorDetails: [ { _id: 2, name: 'Author Two' } ]
  },
  {
    _id: ObjectId("65fd9401b650b778080af418"),
    title: 'Post Title 4',
    body: 'Body of post.',
    category: 'Event',
    likes: 4,
    tags: [ 'news', 'events' ],
    date: 'Fri Mar 22 2024 19:51:53 GMT+0530 (India Standard Time)',
    authorId: 1,
    authorDetails: [ { _id: 1, name: 'Author One' } ]
  }
]
```

Aggregation Pipeline

7. out

```
Atlas atlas-ynt2d2-shard-0 [primary] test> db.posts.aggregate([
...   {
...     $group: {
...       _id: "$category",  // Group by the "category" field
...       numberOfPosts: { $sum: 1 }  // Count the number of posts in each category
...     }
...   },
...   {
...     $out: "aggregatedPosts"  // Writes the result to "aggregatedPosts" collection
...   }
... ])
```

Replication

- Replication is the process of synchronizing data across multiple servers
- Replication in MongoDB provides redundancy and high availability, and is the basis for all production deployment
- It works by maintaining multiple copies of data across separate database servers.
- Replication protects a database from the loss of a single server.
- Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

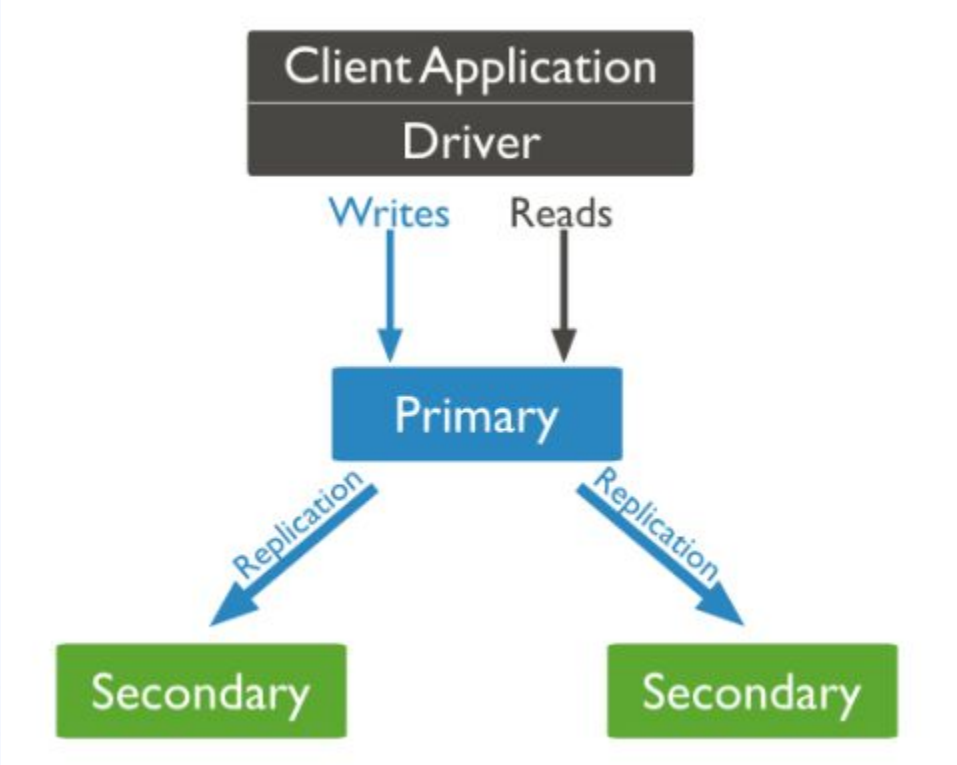
Why Replication?

- To keep your data safe
- High (24*7) availability of data
- Disaster recovery
- No downtime for maintenance (like backups, index rebuilds, compaction)
- Read scaling (extra copies to read from)
- Replica set is transparent to the application

How Replication Works in MongoDB

- MongoDB achieves replication by the use of replica set
- A replica set in MongoDB is a group of **mongod** processes that maintain the same data set
- **mongod** is the main program that runs MongoDB. It's responsible for managing the database, handling requests from applications, and doing other essential task.
- In a replica set, one node is primary node and remaining nodes are secondary. The primary node is the only member in the replica set that receives write operations.

How Replication Works in MongoDB



MongoDB - Relationships

- Relationships in MongoDB represent how various documents are logically related to each other.
- Two approaches
  - Embedded Approach
  - Referenced Approach

Embedded Relationships

Following is the sample document structure of user document

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "name": "Tom Hanks",
   "contact": "987654321",
   "dob": "01-01-1991"
}
```

Following is the sample document structure of address document

```
{
   "_id":ObjectId("52ffc4a5d85242602e000000"),
   "building": "22 A, Indiana Apt",
   "pincode": 123456,
   "city": "Los Angeles",
   "state": "California"
}
```

Embedded Relationships Contd

```
db.users.insert({
          "contact": "987654321",
          "dob": "01-01-1991",
          "name": "Tom Benzamin",
          "address": [
               {
                    "building": "22 A, Indiana Apt",
                    "pincode": 123456,
                    "city": "Los Angeles",
                    "state": "California"
               },
               {
                    "building": "170 A, Acropolis Apt",
                    "pincode": 456789,
                    "city": "Chicago",
                    "state": "Illinois"
               }
          ]
})
```

The whole document can be retrieved in a single query such as

```
>db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

Referenced Relationships

```
// Insert a user
db.users.insert({
  username: "john_doe",
  email: "john@example.com"
})

// Insert another user
db.users.insert({
  username: "jane_smith",
  email: "jane@example.com"
})

// Display users
db.users.find()
```

```
// Insert a post referencing the first user
db.posts.insert({
  title: "Post 1",
  content: "This is the content of post 1",
  user_id: db.users.findOne({ username: "john_doe" })._id
})

// Insert another post referencing the second user
db.posts.insert({
  title: "Post 2",
  content: "This is the content of post 2",
  user_id: db.users.findOne({ username: "jane_smith" })._id
})

// Display posts
db.posts.find()
```

Referenced Relationships ctd.

```
db.users.aggregate([
  {
    $lookup: {
      from: "posts",
      localField: "_id",
      foreignField: "user_id",
      as: "user_posts"
    }
  }
])
```

Database References

- In cases where a document contains references from different collections, we can use MongoDB DBRefs.
- As an example scenario, where we would use DBRefs instead of manual references,
  - consider a database where we are storing different types of addresses (home, office, mailing, etc.) in different collections (address_home, address_office, address_mailing, etc)
  - Now, when a user collection's document references an address, it also needs to specify which collection to look into based on the address type.
  - In such scenarios where a document references documents from many collections, we should use DBRefs.

Database References ctd.

Fields in DB ref,

- $ref - This field specifies the collection of the referenced document
- $id - This field specifies the _id field of the referenced document
- $db - This is an optional field and contains the name of the database in which the referenced document lies

Database References ctd.

```
{
   "_id":ObjectId("53402597d852426020000002"),
   "address": {
   "$ref": "address_home",
   "$id": ObjectId("534009e4d852427820000002"),
   "$db": "tutorialspoint"},
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin"
}
```

Database References ctd.

```
>var user = db.users.findOne({"name":"Tom Benzamin"})
>var dbRef = user.address
>db[dbRef.$ref].findOne({"_id":(dbRef.$id)})
```

```
{
    "_id" : ObjectId("534009e4d852427820000002"),
    "building" : "22 A, Indiana Apt",
    "pincode" : 123456,
    "city" : "Los Angeles",
    "state" : "California"
}
```

Connecting MongoDB with the Backend

- npm install "mongoose"
  - Mongoose is a popular Object Data Modeling (ODM) library for MongoDB in Node.js, and it offers several advantages over using plain boilerplate MongoDB queries.

Advantages of Mongoose

- Schema Definition
- Simplified Queries
- Validation
- Middleware
- Built-in Promises
- Data Type Casting
- Plugins and Hooks

Connecting MongoDB with the Backend

- Create a new folder "Database"
- Create a new file as "Connect.js"

server > Database > JS connect.js > ...

```
1   const mongoose = require('mongoose');
2
3   const connectDB = (url) =>{
4       mongoose.set('strictQuery', true);
5
6       mongoose.connect(url)
7           .then(()=>console.log("MongoDB connected"))
8           .catch((error)=>console.log(error));
9   }
10
11
12
13  module.exports = connectDB;
```

MongoDB Connection Ctd.

1. `const mongoose = require('mongoose');`
    - This line imports the `mongoose` module, which is an Object Data Modeling (ODM) library for MongoDB and provides a convenient way to interact with MongoDB databases using JavaScript.
2. `const connectDB = (url) => {`
    - This line declares a function named `connectDB` that takes a parameter `url`. This function is responsible for establishing a connection to a MongoDB database using the provided URL.
3. `mongoose.set('strictQuery', true);`
    - This line sets the Mongoose option `strictQuery` to `true`. When `strictQuery` is enabled, Mongoose will throw an error if a query contains undefined fields.
4. `mongoose.connect(url)`
    - This line initiates the connection to the MongoDB database using the provided `url`. It establishes a connection to the MongoDB server and returns a promise.

MongoDB Connection Ctd.

5. `.then(() => console.log("MongoDB connected"))`
    - This line registers a callback function that is executed when the `mongoose.connect()` promise is resolved successfully. In this case, it logs the message "MongoDB connected" to the console, indicating that the connection to the MongoDB database was successful.
6. `.catch((error) => console.log(error));`
    - This line registers a callback function that is executed when the `mongoose.connect()` promise is rejected or encounters an error. It logs the error message to the console, providing information about the error that occurred during the connection process.
7. `module.exports = connectDB;`
    - This line exports the `connectDB` function, making it available to other modules when they require or import this module. It allows other parts of the application to use this function to establish a connection to the MongoDB database.

MongoDB Connection Ctd.

- Create .env file

server > .env

```
1   MONGODB_URL = mongodb+srv://MERN:Mern123@cluster0.54hzvs3.mongodb.net/?retryWrites=true&w=majority
```

MongoDB Connection Ctd.

- Update server.js (index) file

```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const connectDB = require('./Database/connect');

require('dotenv').config();
connectDB(process.env.MONGODB_URL);

const app = express();
const PORT = 5000;
```

Validations in Database Operations

When you define a schema in Mongoose, you can specify various validation constraints on the fields, and Mongoose will ensure that these constraints are met before the data is saved to the database.

Type validation:
By specifying a field type (e.g., String, Number), Mongoose will ensure that the data being saved matches the expected type.

Required validation:
required: true
This will ensure that the field is provided before saving the document.

Validations in Database Operations

Unique validation:
unique: true
This ensures that the value is unique across all documents in the collection
Number-specific validators:
min and max: Set the minimum and maximum allowed values for number fields.
String-specific validators:

- minlength and maxlength: These set the minimum and maximum lengths for string values, respectively.
- enum: Allows you to specify an array of valid string values for the field.

Validations in Database Operations

Match pattern of a regular expression:
Eg: match: /^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$/

- ^: Matches the start of a line.
- [\w-\.]+: Matches one or more word characters (\w), hyphens (-), or periods (.). This covers the local part of an email, which appears before the @ symbol. For example, in the email john.doe@example.com, john.doe is the local part.
- @: Matches the @ symbol.
- ([\w-]+\.)+: Matches one or more word characters or hyphens followed by a period (.). This is placed inside a group and followed by a + to allow for domain subdomains. For example, in the domain mail.example.com, mail. is a subdomain.

Validations in Database Operations

Match pattern of a regular expression:

Eg: match: /^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$/

- [\w-]{2,4}: Matches between 2 to 4 word characters or hyphens. This covers top-level domains (TLDs) like .com, .net, .org, but may not include newer or longer TLDs like .design or .solutions.
- $: Matches the end of a line.

Custom validation:

Eg:

```
validate: {
    validator: function(value) {
        return value >= 18;
    },
    message: 'Age must be at least 18.'
}
```

Creating Database Schema Model

- Data in MongoDB has a flexible schema. Collections do not enforce document structure by default.
- Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's collections, by default, do not require their documents to have the same schema. That is:
  - The documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- In practice, however, the documents in a collection share a similar structure, and you can enforce document validation rules for a collection during update and insert operations.

Schema Validation

- Schema validation ensures data consistency and integrity within the database.
- It helps prevent invalid or inconsistent data from being saved to the database.
- Validation rules are defined within the schema, reducing the need for manual checks in your application code.

Key Features of Schema Validation

- **Data Types**: Define the expected data types for each field (e.g., String, Number, Date).
- **Required Fields**: Specify which fields must have values before saving a document.
- **Custom Validation Functions**: Implement custom validation logic for specific fields.
- **Enum Validation**: Restrict a field to a predefined set of values.
- **Min/Max Values**: Set minimum and maximum values for numeric fields.
- **Regular Expressions**: Use regular expressions to validate string formats.
- **Error Messages**: Customize error messages for validation failures.

```
const productSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    unique: true,
    trim: true,
  },
  price: {
    type: Number,
    required: true,
    min: 0.01,
  },
  category: {
    type: String,
    enum: ['Electronics', 'Clothing', 'Furniture'],
  },
});
```

Creating Database Schema Model

server > Database > models > JS messages.js > ...

```
1   const mongoose = require('mongoose');
2
3   const MessageSchema = new mongoose.Schema({
4       message:{
5           type:String,
6           required:true
7       }
8   });
9
10  const Message = mongoose.model('Message', MessageSchema);
11
12  module.exports = Message;
```

Creating Database Schema Model ctd.

- The **mongoose.model()** function of the mongoose module is used to create a collection of a particular database of MongoDB.

Creating APIs - Post Message

server > routes > JS postmessage.route.js > ...

```
1   const express = require('express');
2   const Message = require('../Database/models/messages');
3
4   const router = express.Router();
5
6   router.post('/',async(req,res)=>{
7       try{
8           const message = new Message(req.body);
9           await message.save();
10          res.status(200).json({
11              status: 'success',
12              data:{
13                  message
14              }
15          })
16      }catch(err){
17          res.status(500).json({
18              status:'Failed',
19              message: (err)
20      });
21      }
22
23  })
24
25  module.exports = router
```

Creating APIs - Get Message

server > routes > JS getmessage.route.js > <unknown>

```
1   const express = require('express');
2   const Message = require('../Database/models/messages');
3
4   const router = express.Router();
5
6   router.get('/',async(req,res)=>{
7       const messages = await Message.find({})
8       try{
9           res.status(200).json({
10              status:'success',
11              data:{
12                  messages
13              }
14          })
15      }catch(err){
16          res.status(500).json({
17              staus:'Failed',
18              message: err
19      })
20      }
21  })
22
23  module.exports = router
```

Creating APIs - Get Message by ID

server > routes > JS getmessagebyid.route.js > <unknown>

```
1  const express = require('express');
2  const Message = require('../Database/models/messages');
3
4  const router = express.Router();
5
6  router.get('/:id',async (req,res)=>{
7
8      const message = await Message.findById(req.params.id);
9      try{
10         res.status(200).json({
11             status: 'success',
12             data:{
13                 message
14             }
15         })
16     }catch(err){
17         res.status(500).json({
18             status:'Failes',
19             message: err
20         });
21     }
22 })
23
24 module.exports = router
```

Creating Database Schema Model

server > Database > models > JS messages.js > ...

```
1   const mongoose = require('mongoose');
2
3   const MessageSchema = new mongoose.Schema({
4       message:{
5           type:String,
6           required:true
7       }
8   });
9
10  const Message = mongoose.model('Message', MessageSchema);
11
12  module.exports = Message;
```

Creating Database Schema Model ctd.

- The **mongoose.model()** function of the mongoose module is used to create a collection of a particular database of MongoDB.

Creating APIs - Post Message

server > routes > JS postmessage.route.js > ...

```
1   const express = require('express');
2   const Message = require('../Database/models/messages');
3
4   const router = express.Router();
5
6   router.post('/',async(req,res)=>{
7       try{
8           const message = new Message(req.body);
9           await message.save();
10          res.status(200).json({
11              status: 'success',
12              data:{
13                  message
14              }
15          })
16      }catch(err){
17          res.status(500).json({
18              status:'Failed',
19              message: (err)
20      });
21      }
22
23  })
24
25  module.exports = router
```

Creating APIs - Get Message

server > routes > JS getmessage.route.js > <unknown>

```
1   const express = require('express');
2   const Message = require('../Database/models/messages');
3
4   const router = express.Router();
5
6   router.get('/',async(req,res)=>{
7       const messages = await Message.find({})
8       try{
9           res.status(200).json({
10              status:'success',
11              data:{
12                  messages
13              }
14          })
15      }catch(err){
16          res.status(500).json({
17              staus:'Failed',
18              message: err
19      })
20      }
21  })
22
23  module.exports = router
```

Creating APIs - Get Message by ID

server > routes > JS getmessagebyid.route.js > <unknown>

```
1  const express = require('express');
2  const Message = require('../Database/models/messages');
3
4  const router = express.Router();
5
6  router.get('/:id',async (req,res)=>{
7
8      const message = await Message.findById(req.params.id);
9      try{
10         res.status(200).json({
11             status: 'success',
12             data:{
13                 message
14             }
15         })
16     }catch(err){
17         res.status(500).json({
18             status:'Failes',
19             message: err
20         });
21     }
22 })
23
24 module.exports = router
```

Creating APIs - Update Message

```
const express = require('express');
const Message = require('../Database/models/messages');

const router = express.Router();

router.put('/:id',async (req,res) => {
    try {
        const updateMessage = await Message.findByIdAndUpdate(
          req.params.id,
          { message: req.body.newMessage },
          { new: true, runValidators: true }
        );

        res.status(200).json({
          status: 'success',
          data: {
            updateMessage: updateMessage
          }
        });
      } catch (err) {
        res.status(500).json({
          status: 'Failed',
          message: err
        });
      }
```

Creating APIs - Delete Message

server > routes > JS deletemessage.route.js > router.delete('/:id') callback

```
1   const express = require('express');
2   const Message = require('../Database/models/messages');
3
4   const router = express.Router();
5
6   router.delete('/:id', async (req,res)=>{
7       try{
8           const deleteMessage = await Message.findByIdAndRemove(req.params.id, req.body);
9
10          res.status(200).json({
11              status: 'success',
12              msg: deleteMessage
13          });
14      }catch(err){
15          res.status(500).json({
16              status:'Failed',
17              msg: err
18          });
19      }
20  })
21
22  module.exports = router;
```

Using Routes/APIs

server > JS server.js > ...

```
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  const cors = require('cors');
4  const connectDB = require('./Database/connect');
5
6  const postMessageRoute = require('./routes/postmessage.route');
7  const getMessageRoute = require('./routes/getmessage.route');
8  const updateMessageRoute = require('./routes/updatemessage.route');
9  const deleteMessageRoute = require('./routes/deletemessage.route');
10 const getSpecificRoute = require('./routes/getmessagebyid.route');
11
12 require('dotenv').config();
13 connectDB(process.env.MONGODB_URL);
14 const app = express();
15 const PORT = 5000;
16
17 app.use(cors());
18 app.use(bodyParser.json());
19
20 app.listen(PORT, () => {
21     console.log(`Server is running on http://localhost:${PORT}`);
22 });
23
24 app.use('/post-message',postMessageRoute);
25 app.use('/get-message',getMessageRoute);
26 app.use('/update-message',updateMessageRoute);
27 app.use('/delete-message',deleteMessageRoute);
28 app.use('/get-message',getSpecificRoute);
```

Thanks!